Programare orientată pe obiecte

Laboratorul 7

Mihai Nan
mihai.nan.cti@gmail.com

Facultatea de Automatică şi Calculatoare
Universitatea Politehnica din Bucureşti
Anul universitar 2020 - 2021

1 Colectii

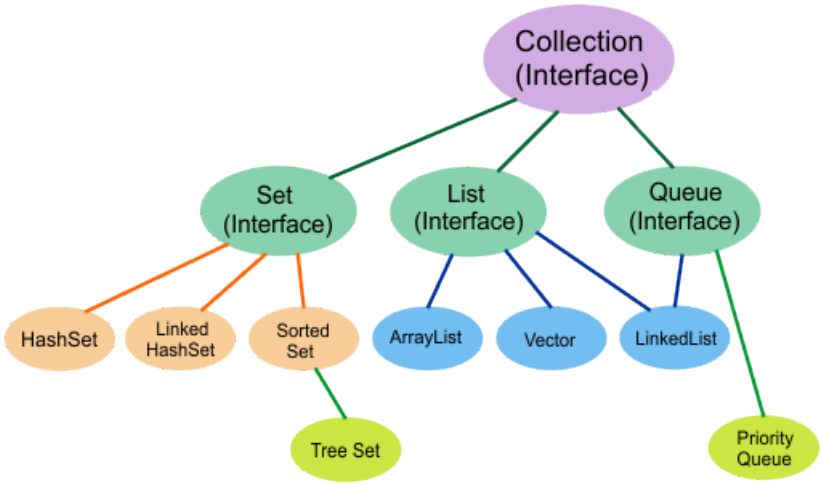
1.1 Introducere

O **colectie** este un obiect care grupeaza mai multe elemente intr-o singura unitate. Prin intermediul colectiilor, vom avea acces la diferite tipuri de structuri de date cum ar fi vectori, liste inlantuite, stive, multimi matematice, tabele de dispersie, etc. Colectiile sunt folosite atat pentru memorarea si manipularea datelor, cat si pentru transmiterea unor informatii de la o metoda la alta.

⚠ IMPORTANT !

⚠ Tipul de date al elementelor dintr-o colectie este ***Object***, ceea ce inseamna ca multimile reprezentate sunt eterogene, putand include obiecte de orice tip.

Clasele si interfetele pe care le ofera limbajul Java, in vederea lucrului cu colectii de obiecte, se afla in pachetul ***java.util***. In consecinta, pentru a le putea folosi usor, trebuie sa folosim clauze import corespunzatoare.



Interfetele reprezinta nucleul mecanismului de lucru cu colectii, scopul lor fiind de a permite utilizarea structurilor de date independent de modul lor de implementare. Toate clasele concrete care au ca supertip interfata ***Collection*** implementeaza conceptul de colectie de obiecte.

1.2 Interfata Collection

Collection modeleaza o colectie la nivelul cel mai general, descriind un grup de obiecte numite si elementele sale. Unele implementari ale acestei interfete permit existenta elementelor duplicate, alte implementari nu. Unele au elementele ordonate, altele nu. Platforma Java nu ofera o implementare directa a acestei interfete, ci exista doar implementari ale unor subinterfete mai concrete, cum ar fi ***Set*** sau ***List***.

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
Hashtable	x			No	No
TreeMap	x			Sorted	By *natural order* or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By *natural order* or custom comparison rules
LinkedHashSet		x		By insertion order	No
ArrayList			x	By index	No
Vector			x	By index	No
LinkedList			x	By index	No
PriorityQueue				Sorted	By to-do order

1.2.1 Liste

Interfata ***List***, pe langa metodele mostenite de la interfata ***Collection***, mai are si alte metode proprii. Printre clasele predefinite care implementeaza interfata ***List*** sunt ***ArrayList*** si ***LinkedList***.

Clasa ***LinkedList*** furnizeaza o implementare a interfetei ***List***, elementele propriu-zisa ale listei fiind stocate sub forma unei liste inlantuite. Acest fapt asigura un timp mai bun pentru stergerea si inserarea unui element in interiorul listei comparativ cu ***ArrayList***. In schimb, accesul aleator la un element din interiorul listei este o operatie consumatoare de mai mult timp fata de ***ArrayList***.

Clasa ***LinkedList*** are cateva metode in plus, fata de cele din interfata ***List***, iar acestea permit prelucrarea elementelor aflate la cele doua capete ale listei.

Pentru o mai buna intelegere a claselor care implementeaza aceasta interfata, se recomanda analizarea exemplului de mai jos.

Cod sursa Java

```
class Lista {
    List list1;
    List list2;

    public Lista() {
        this.list1 = new ArrayList();
        this.list2 = new LinkedList();
    }

    public static void main(String args[]) {
        Lista obj = new Lista();
        obj.list1.add("Lab POO");
        obj.list1.add("Colectii");
        obj.list1.add("Structuri de date");
        if(obj.list1.contains("Colectii")) {
            System.out.println("Lista contine cuvantul");
        }
        int index = 0;
        while(!obj.list1.isEmpty()) {
            System.out.println(obj.list1.get(index));
            obj.list1.remove(index);
        }
        obj.list2.add(1);
        obj.list2.add(10);
        obj.list2.add(20);
        Vector v = new Vector();
        v.add(50);
        v.add(17);
        obj.list2.addAll(v);
        for(Object i : obj.list2) {
            System.out.println(i);
        }
```

```
33            Collections.sort(obj.list2);
34            System.out.println(obj.list2);
35        }
36    }
```

1.2.2 Multimi

Set modeleaza notiunea de multime in sens matematic. O multime nu poate avea elemente duplicate, mai bine zis, nu poate contine doua obiecte ***o1*** si ***o2*** cu proprietatea ca ***o1.equals(o2)***. Aceasta mosteneste metodele din ***Collection***, fara a avea alte metode specifice. Doua dintre clasele standard care ofera implementari concrete ale acestei interfete sunt ***HashSet*** si ***TreeSet***.

SortedSet este asemanatoare cu interfata ***Set***, diferenta principala constand in faptul ca elementele dintr-o astfel de colectie sunt ordonate ascendent. Aceasta pune la dispozitie operatii care beneficiaza de avantajul ordonarii elementelor. Ordonarea elementelor se face conform ordinii lor naturale, sau conform cu ordinea data de un comparator specificat la crearea colectiei si este mentinuta automat la orice operatie efectuata asupra multimii.

⚠ IMPORTANT !

> ⚠ In cazul unei colectii de tip ***SortedSet***, pentru orice doua obiecte ***o1***, ***o2*** ale colectiei, apelul ***o1.compareTo(o2)*** (sau ***comparator.compare(o1, o2)***, daca este folosit un comparator) trebuie sa fie valid si sa nu provoace exceptii.

Cod sursa Java

```
1  public interface SortedSet extends Set {
2      //Subliste
3      SortedSet subSet(Object fromElement, Object toElement);
4      SortedSet headSet(Object toElement);
5      SortedSet tailSet(Object fromElement);
6
7      //Capete
8      Object first();
9      Object last();
10
11     Comparator comparator();
12 }
```

1.3 Dictionare

Map descrie structuri de date ce asociaza fiecarui element o cheie unica, dupa care poate fi regasit. Obiectele de acest tip nu pot contine chei duplicate si fiecare cheie este asociata la un singur element. Ierarhia interfetelor derivate din ***Map*** este independenta de ierarhia derivata din ***Collection***. Clase care implementeaza interfata ***Map*** sunt ***HashMap***, ***TreeMap*** si ***Hashtable***.

SortedMap este asemanatoare cu interfata ***Map***, la care se adauga faptul ca multimea cheilor dintr-o astfel de colectie este mentinuta ordonata ascendent, conform ordinii naturale, sau conform cu ordinea data de un comparator specificat la crearea colectiei. Este subclasa a interfetei ***Map***, oferind metode suplimentare pentru: extragere de subtabele, aflarea primei / ultimei chei, aflarea comparatorului folosit pentru ordonare.

1.4 Iteratori si enumerari

Enumerarile si iteratorii descriu modalitati pentru parcurgerea secventiala a unei colectii, indiferent daca aceasta este idexata sau nu. Ei sunt descrisi de obiecte ce implementeaza interfetele ***Enumeration***, respectiv ***Iterator***

sau ***ListIterator***.

1.4.1 Enumeration

Cod sursa Java

```
class TestEnumeration {
    public static void main(String args[]) {
        Vector v = new Vector();
        for(int i = 0; i < 10; i++) {
            v.add((int) (Math.random()*100));
        }
        //Parcurgerea elementelor din vector
        Enumeration en = v.elements();
        while(en.hasMoreElements()) {
            int x = (int) en.nextElement();
            System.out.println(x);
        }
    }
}
```

1.4.2 Iterator

Cod sursa Java

```
class TestIterator {
    public static void main(String args[]) {
        Vector v = new Vector();
        for(int i = 0; i < 10; i++) {
            v.add((int) (Math.random()*100));
        }
        //Parcurgerea elementelor din vector
        //si stergerea celor nule
        for(Iterator itr = v.iterator(); itr.hasNext();) {
            int x = (int) itr.next();
            if(x == 0)
                itr.remove();
        }
    }
}
```

1.4.3 ListIterator

Cod sursa Java

```
class TestListIterator {
    public static void main(String args[]) {
        Vector v = new Vector();
        for(int i = 0; i < 10; i++) {
            v.add((int) (Math.random()*100));
        }
        //Parcurgerea elementelor din vector
        //si inlocuirea celor nule
        for(ListIterator i=v.listIterator(); i.hasNext();) {
            int x = (int) i.next();
            if(x == 0) {
                i.set((int) 10);
            }
        }
```

```
    }
}
```

⚠ IMPORTANT !

⚠ Deoarece colectiile sunt construite peste tipul de date ***Object***, metodele de tip ***next*** sau ***prev*** ale iteratorilor vor returna tipul ***Object***, fiind responsabilitatea noastra de a face conversie la alte tipuri de date, daca este cazul.

Observatie

Iteratorii simpli permit eliminarea elementului curent din colectia pe care o parcurg, iar cei de tip ***ListIterator*** permit si inserarea unui element la pozitia curenta, respectiv modificarea elementului curent, precum si iterarea in ambele sensuri.

2 Genericitate

Pentru intelegerea nevoii de de abstractizare a tipurilor de date, vom urmari exemplu de mai jos si vom trasa principalele dezavantaje ale utilizarii colectiilor fara acest mecanism.

Cod sursa Java

```
List list = new ArrayList();
list.add(5);
int x = (int) list.iterator().next();
list.add("Text");
String text = (String) list.get(1);
```

In exemplul prezentat, se observa necesitatea operatiei de *cast* pentru a identifica corect variabila obtinuta din lista. Principalele dezavantaje, intalnite intr-o astfel de abordare, sunt: citirea codului este ingreunata, apare posibilitatea unor erori de executie, in momentul in care intr-o colectie ordonata se introduc obiecte de tipuri diferite, care nu se pot compara.

Genericitatea intervine tocmai pentru a elimina aceste probleme.

Cod sursa Java

```
List<Integer> list = new ArrayList<Integer>();
list.add(5);
list.add(7);
int x = list.iterator().next();
int y = list.get(1);
```

In aceasta situatie, lista nu mai poate contine obiecte oarecare, ci poate contine **doar** obiecte de tipul ***Integer***. In plus, se observa ca se elimina si *cast*-ul. De aceasta data, verificarea tipurilor este efectuata de **compilator**, ceea ce elimina potentialele erori de executie cauzate de eventuale *cast*-uri incorecte.

Beneficiile dobandite prin utilizarea acestui mecanism sunt: imbunatatirea **lizibilitatii** codului, cresterea gradului de **robustete**.

3 *equals()* vs *hashcode()*

Metoda ***hashCode()*** returneaza un intreg care reprezinta hashcodul pentru obiectul respectiv. Un **hashcode** este o valoare de control pentru un obiect, fiind o valoare aleatoare generata pe baza continutului obiectului. **Hashcodul** este diferit pentru instante din aceasi clasa, dar cu continut diferit si trebuie sa fie acelasi pentru obiecte cu acelasi continut. Este utilizat pentru stocarea obiectelor intr-un ***Hashtable***, implementarea metodei ***hashCode()*** din clasa ***Object*** atasand fiecarei instante o valoare unica.

Metoda ***equals()***, dupa cum stim deja, este uzitata pentru a verifica egalitatea a doua obiecte. Este important de remarcat faptul ca, in cazul in care se doreste suprascrierea acestei metode pentru o anumita clasa, se recomanda si suprascrierea metodei ***hashCode()***, pentru a nu se ajunge la comportamente paradoxale, in cazul uzitarii unor colectii. In acest sens, se recomanda analizarea codului propus.

Cod sursa Java

```
class Student {
    private String nume;
    private double medie;
    public Student(String nume, double medie) {
        this.nume = nume;
        this.medie = medie;
    }
    public boolean equals(Object o) {
        Student s = (Student) o;
        if(s.medie == this.medie) {
            return true;
        } else {
            return false;
        }
    }
    public String toString() {
        String result = this.nume + " - " + this.medie;
        return result;
    }
    public static void main(String args[]) {
        ArrayList<Student> list = new ArrayList<Student>();
        HashSet<Student> set = new HashSet<Student>();
        set.add(new Student("Popescu", 9.75));
        list.add(new Student("Popescu", 9.75));
        Student s = new Student("Ionescu", 9.35);
        set.add(new Student("Ionescu", 9.35));
        list.add(new Student("Ionescu", 9.35));
        System.out.println("Test HashSet");
        if(set.contains(s)) {
            System.out.println("A fost adaugat!");
        } else {
            System.out.println("Nu a fost adaugat!");
            System.out.println(s);
            System.out.println(set);
        }
        System.out.println("Test ArrayList");
        if(list.contains(s)) {
            System.out.println("A fost adaugat!");
            System.out.println(s);
            System.out.println(list);
        }
    }
}
```